Introduction to Access Control

We can now start using CrypTen to carry out private computations in some common use cases. In this tutorial, we will demonstrate how CrypTen would apply for the scenarios described in the Introduction. In all scenarios, we'll use a simple two-party setting and demonstrate how we can learn a linear SVM. In the process, we will see how access control works in CrypTen.

As usual, we'll begin by importing the `crypten` and `torch` libraries, and initialize `crypten` with `crypten.init()`.

In [1]:
```
import crypten
import torch

crypten.init()
torch.set_num_threads(1)
```

Setup

In this tutorial, we will train a Linear SVM to perform binary classification. We will first generate 1000 ground truth samples using 100 features and a randomly generated hyperplane to separate positive and negative examples.

(Note: this will cause our classes to be linearly separable, so a linear SVM will be able to classify with perfect accuracy given the right parameters.)

We will also include a test set of examples (that are also linearly separable by the same hyperplane) to show that the model learns a general hyperplane rather than memorizing the training data.

In [2]:
```
num_features = 100
num_train_examples = 1000
num_test_examples = 100
epochs = 40
lr = 3.0

# Set random seed for reproducibility
torch.manual_seed(1)

features = torch.randn(num_features, num_train_examples)
w_true = torch.randn(1, num_features)
b_true = torch.randn(1)

labels = w_true.matmul(features).add(b_true).sign()

test_features = torch.randn(num_features, num_test_examples)
test_labels = w_true.matmul(test_features).add(b_true).sign()
```

Now that we have generated our dataset, we will train our SVM in four different access control scenarios across two parties, Alice and Bob:

- Data Labeling: Alice has access to features, while Bob has access to labels
- Feature Aggregation: Alice has access to the first 50 features, while Bob has access to the last 50 features
- Data Augmentation: Alice has access to the first 500 examples, while Bob has access to the last 500 examples
- Model Hiding: Alice has access to `w_true` and `b_true`, while Bob has access to data samples to be classified

Throughout this tutorial, we will assume Alice is using the rank 0 process, while Bob is using the rank 1 process. Additionally we will initialize our weights using random values.

In [3]:
```
ALICE = 0
BOB = 1
```

In each example, we will use the same code to train our linear SVM once the features and labels are properly encrypted. This code is contained in `examples/mpc_linear_svm`, but it is unnecessary to understand the training code to properly use access control. The training process itself is discussed in depth in later tutorials.

In [4]:
```
from examples.mpc_linear_svm.mpc_linear_svm import train_linear_svm, eva
```

```
---------------------------------------------------------------------------
-----
ModuleNotFoundError                       Traceback (most recent call
 last)
<ipython-input-4-7314d3ab9618> in <module>
----> 1 from examples.mpc_linear_svm.mpc_linear_svm import train_linea
r_svm, evaluate_linear_svm

ModuleNotFoundError: No module named 'examples.mpc_linear_svm'
```

Saving / Loading Data

We have now generated features and labels for our model to learn. In the scenarios we explore in this tutorial, we would like to ensure that each party only has access to some subset of the data we have generated. To do so, we will use special save / load methods that CrypTen provides to handle loading only to a specified party and synchronizing across processes.

We will use `crypten.save()` here to save data from a particular source, then we will load using `crypten.load()` in each example to load on a particular source. The following code will save all data we will use to files, then each example will load its data as necessary.

(Note that because we are operating on a single machine, all processes will have access to all of the files we are using. However, this still will work as expected when operating across machines.)

In [ ]:
```
from crypten import mpc

# Specify file locations to save each piece of data
filenames = {
    "features": "/tmp/features.pth",
    "labels": "/tmp/labels.pth",
    "features_alice": "/tmp/features_alice.pth",
    "features_bob": "/tmp/features_bob.pth",
    "samples_alice": "/tmp/samples_alice.pth",
    "samples_bob": "/tmp/samples_bob.pth",
    "w_true": "/tmp/w_true.pth",
    "b_true": "/tmp/b_true.pth",
    "test_features": "/tmp/test_features.pth",
    "test_labels": "/tmp/test_labels.pth",
}


@mpc.run_multiprocess(world_size=2)
def save_all_data():
    # Save features, labels for Data Labeling example
    crypten.save(features, filenames["features"])
    crypten.save(labels, filenames["labels"])

    # Save split features for Feature Aggregation example
    features_alice = features[:50]
    features_bob = features[50:]

    crypten.save(features_alice, filenames["features_alice"], src=ALICE)
    crypten.save(features_bob, filenames["features_bob"], src=BOB)

    # Save split dataset for Dataset Aggregation example
    samples_alice = features[:, :500]
    samples_bob = features[:, 500:]
    crypten.save(samples_alice, filenames["samples_alice"], src=ALICE)
    crypten.save(samples_bob, filenames["samples_bob"], src=BOB)

    # Save true model weights and biases for Model Hiding example
    crypten.save(w_true, filenames["w_true"], src=ALICE)
    crypten.save(b_true, filenames["b_true"], src=ALICE)

    crypten.save(test_features, filenames["test_features"], src=BOB)
    crypten.save(test_labels, filenames["test_labels"], src=BOB)

save_all_data()
```

Scenario 1: Data Labeling

Our first example will focus on the *Data Labeling* scenario. In this example, Alice has access to features, while Bob has access to the labels. We will train our linear svm by encrypting the features from Alice and the labels from Bob, then training our SVM using an aggregation of the encrypted data.

In order to indicate the source of a given encrypted tensor, we encrypt our tensor using `crypten.load()` (from a file) or `crypten.cryptensor()` (from a tensor) using a keyword argument `src`. This `src` argument takes the rank of the party we want to encrypt from (recall that ALICE is 0 and BOB is 1).

(If the `src` is not specified, it will default to the rank 0 party. We will use the default when encrypting public values since the source is irrelevant in this case.)

In [ ]:
```
from crypten import mpc

@mpc.run_multiprocess(world_size=2)
def data_labeling_example():
    """Apply data labeling access control model"""
    # Alice loads features, Bob loads labels
    features_enc = crypten.load(filenames["features"], src=ALICE)
    labels_enc = crypten.load(filenames["labels"], src=BOB)

    # Execute training
    w, b = train_linear_svm(features_enc, labels_enc, epochs=epochs, lr=

    # Evaluate model
    evaluate_linear_svm(test_features, test_labels, w, b)

data_labeling_example()
```

Scenario 2: Feature Aggregation

Next, we'll show how we can use CrypTen in the *Feature Aggregation* scenario. Here Alice and Bob each have 50 features for each sample, and would like to use their combined features to train a model. As before, Alice and Bob wish to keep their respective data private. This scenario can occur when multiple parties measure different features of a similar system, and their measurements may be proprietary or otherwise sensitive.

Unlike the last scenario, one of our variables is split among two parties. This means we will have to concatenate the tensors encrypted from each party before passing them to the training code.

In [ ]:
```
@mpc.run_multiprocess(world_size=2)
def feature_aggregation_example():
    """Apply feature aggregation access control model"""
    # Alice loads some features, Bob loads other features
    features_alice_enc = crypten.load(filenames["features_alice"], src=A
    features_bob_enc = crypten.load(filenames["features_bob"], src=BOB)

    # Concatenate features
    features_enc = crypten.cat([features_alice_enc, features_bob_enc], c

    # Encrypt labels
    labels_enc = crypten.cryptensor(labels)

    # Execute training
    w, b = train_linear_svm(features_enc, labels_enc, epochs=epochs, lr=

    # Evaluate model
    evaluate_linear_svm(test_features, test_labels, w, b)

feature_aggregation_example()
```

Scenario 3: Dataset Augmentation

The next example shows how we can use CrypTen in a *Data Augmentation* scenario. Here Alice and Bob each have 500 samples, and would like to learn a classifier over their combined sample data. This scenario can occur in applications where several parties may each have access to a small amount of sensitive data, where no individual party has enough data to train an accurate model.

Like the last scenario, one of our variables is split amongst parties, so we will have to concatenate tensors from encrypted from different parties. The main difference from the last scenario is that we are concatenating over the other dimension (the sample dimension rather than the feature dimension).

In [ ]:
```
@mpc.run_multiprocess(world_size=2)
def dataset_augmentation_example():
    """Apply dataset augmentation access control model"""
    # Alice loads some samples, Bob loads other samples
    samples_alice_enc = crypten.load(filenames["samples_alice"], src=ALI
    samples_bob_enc = crypten.load(filenames["samples_bob"], src=BOB)

    # Concatenate features
    samples_enc = crypten.cat([samples_alice_enc, samples_bob_enc], dim=

    labels_enc = crypten.cryptensor(labels)

    # Execute training
    w, b = train_linear_svm(samples_enc, labels_enc, epochs=epochs, lr=l

    # Evaluate model
    evaluate_linear_svm(test_features, test_labels, w, b)

dataset_augmentation_example()
```

Scenario 4: Model Hiding

The last scenario we will explore involves *model hiding*. Here, Alice has a pre-trained model that cannot be revealed, while Bob would like to use this model to evaluate on private data sample(s). This scenario can occur when a pre-trained model is proprietary or contains sensitive information, but can provide value to other parties with sensitive data.

This scenario is somewhat different from the previous examples because we are not interested in training the model. Therefore, we do not need labels. Instead, we will demonstrate this example by encrypting the true model parameters (`w_true` and `b_true`) from Alice and encrypting the test set from Bob for evaluation.

(Note: Because we are using the true weights and biases used to generate the test labels, we will get 100% accuracy.)

In [ ]:
```
@mpc.run_multiprocess(world_size=2)
def model_hiding_example():
    """Apply model hiding access control model"""
    # Alice loads the model
    w_true_enc = crypten.load(filenames["w_true"], src=ALICE)
    b_true_enc = crypten.load(filenames["b_true"], src=ALICE)

    # Bob loads the features to be evaluated
    test_features_enc = crypten.load(filenames["test_features"], src=BOB

    # Evaluate model
    evaluate_linear_svm(test_features_enc, test_labels, w_true_enc, b_tr

model_hiding_example()
```

In this tutorial we have reviewed four techniques where CrypTen can be used to perform encrypted training / inference. Each of these techniques can be used to facilitate computations

in different privacy-preserving scenarios. However, these techniques can also be combined to increase the amount of scenarios where CrypTen can maintain privacy.

For example, we can combine feature aggregation and data labeling to train a model on data split between three parties, where two parties each have access to a subset of features, and the third party has access to labels.

Before exiting this tutorial, please clean up the files generated using the following code.

In [ ]:
```
import os

for fn in filenames.values():
    if os.path.exists(fn): os.remove(fn)
```